An Object Oriented Architecture

William J. Dally
James T. Kajiya

California Institute of Technology
Pasadena, Ca. 91125

ABSTRACT. We propose a new machine architecture for high performance execution of late binding object oriented languages. The two principal mechanisms for attaining this goal are a fast context allocation/access scheme and an instruction translation lookaside buffer. New ideas in this paper include the concept and implementation of abstract instructions, using floating point addresses to solve the small object problem, and a novel context allocation/access mechanism.

§1 Introduction

The power of object oriented languages derives from a few simple ideas that combine to make a large execution overhead on conventional v.Neumann processors. The purpose of this paper is to describe a number of proposed hardware mechanisms which go some way toward eliminating this overhead.

1.1 Summary

A major execution overhead is *method lookup*. That is, during execution, every single procedure call is made to an *abstract procedure*. An abstract procedure simply consists of a name, the *message* name that must be combined with runtime information to resolve the message name to an actual piece of code, known as a *method*. The method to be executed is found by associating the message name in a hash table for the data type – or *class* – of a selected operand. This association mechanism is quite costly in comparison to the typical overhead for procedure calling in conventional languages. But it has enormous advantages which we outline below. The architectural mechanism which we propose is one which has been used with some success in software implementations. We cache associations into a translation lookaside buffer. We have found through simulations that a rather modest cache results in very high hit ratios. These results indicate that method lookup overhead may be effectively eliminated.

The need to access both great numbers of small objects and a lesser number of large objects is problematical for conventional segmentation schemes. This dilemma is known as the small object problem. We propose the use of floating point addresses to solve this dilemma.

A major execution overhead is context allocation. Each time a message is sent, a general context must be allocated from the heap and have several fields initialized. We propose a novel mechanism for allocating and accessing such contexts.

1.2 Background

To test the architectural features described above we are developing the Caltech Object Machine (COM). The COM draws upon many sources for the ideas in its implementation. Because the cost of context allocation and access for more ordinary languages is not insignificant, a considerable body of work attacking this problem has appeared [8,3,18]. COM is not the first machine to address the execution of Smalltalk with hardware. There are two notable precursors to COM. The first is the Xerox Dorado [6]. While there were no features included in this machine specifically for Smalltalk, considerable attention was given to context allocation. Their microcode kernel Smalltalk implementation is the fastest implementation to date. They have also implemented a method cache in microcode.

The second project is Berkeley's Smalltalk on a RISC (SOAR) project [17]. Several interesting ideas were included in this machine. Words are tagged so that integers may be distinguished from pointers and to support a generation scavenging garbage collector. Contexts are allocated via the RISC register window scheme with a trap for non-LIFO contexts.

Recently, a number of researchers have undertaken Smalltalk software implementation experiments [7,14]. Our work draws heavily on the results of these experiments. Of the software techniques for speeding Smalltalk implementations, the original Smalltalk implementer's guide suggests caching of message hashes. Their caching strategy is direct mapping. The Hewlett-Packard implementation uses a two way set association to great advantage [5,10].

The idea of instruction translation is discussed in [16] where instructions are translated to resolve operand addresses before execution.

1.3 Paper Outline

The next section introduces new architectural features for supporting object oriented languages. Section 3 describes the Caltech Object Machine (COM). Section 4 discusses the mapping of the Smalltalk virtual machine onto COM. Section 5 outlines the results of simulation experiments. The last section closes with a summary and current status of the project.

§2 Architectural Features

2.1 Abstract Instructions

We propose a novel method for interpreting machine instructions. In this scheme a given CPU instruction is, like a virtual address, abstract. The meaning of a particular op code depends upon the type or *Class* of the operand objects of the instruction. Opcode and Class together determine the name of an instruction descriptor, which holds information indicating whether the instruction is primitve or defined.

Since the advantages of this mode of execution are less well known to architecture community than the advantages of the analogous mechanism of virtual addressing, we will briefly outline them here. Instruction safety (run time type checking), a late binding abstraction mechanism (which facilitates the factoring of code), and smooth extensibility (lack of distinction between primitive and nonprimitive code), all stem from an execution mechanism which is essentially equivalent to the Smalltalk message passing paradigm.

The first advantage is *instruction safety* which prevents the all too common occurrence of applying an instruction to the wrong datatype, or attempting to execute data. On a higher level, errors which result from attempting to apply a method to some data structure of inappropriate form can be checked at runtime with no extra execution penalty. The kind of instruction safety built into abstract instructions, however, is more than one that merely checks types and signals errors, but one in which it is impossible to express an erroneous operation: We cannot perform an integer add on a floating number because there is only one token for ADD. The meaning depends on the datatype. In high level languages, this kind of type checking can easily be the principal cost of interpretive overhead. For example, APL spends a great deal of its time simply performing run time typechecks.

The second advantage is *late binding*, an abstraction mechanism key to effective reusability of code. Late binding tries to postpone association (or binding) of a construct and its definition to as late a time as possible: just before execution. The primary reason that algorithms are written and rewritten over and over again is that no one writes general code. Everyone who tries to write general code runs up against an efficiency barrier which forces implementation of less general code than is desirable for reusability. One avenue to generality is late binding. It holds great promise, for, when the execution penalty for writing general code can be collected in one uniform mechanism, it becomes feasible to eliminate the overhead with hardware.

Binding meanings to objects as early as possible is more efficient but less flexible. For example, compiling is more efficient than interpreting, but everyone who has worked under both environments would, if given a choice, always opt for interpretation were it not so slow. In Pascal, the quintessential early binding language, almost everything—even array bounds—must be declared at compile time. This is the primary reason that Pascal feels so rigid and inflexible. It is difficult to write a general sort routine which works on arbitrary length lists of arbitrary types. Early binding forces the code to be so specialized that hope of obtaining a general, reusable code is very slim. In fact, it is not all unusual in Pascal to see multiple versions of codes which save for different type declarations are identical. On the other hand, in Smalltalk, the quintessential late binding language, it is easy to define a general sort routine—one which will even work for lists of datatypes which are not yet defined. Because of this, large databases of reusable code, called *toolkits* in Smalltalk, appear quite commonly. In fact, it is the rule rather than an exception that code is reused—oftentimes in unexpected ways. Unfortunately, wider adoption of late binding is prevented by a severe execution penalty for binding late. Our architecture proposal seeks to eliminate this efficiency barrier. As such, this machine, with only minor differences, would be useful for other languages in which late binding is a prominent feature, most notably APL, Lisp, and Flavors.

The third advantage is, *smooth extensibility* of the architecture. Since each instruction is a token whose meaning is determined in conjunction with the Class of the instruction operand, the exact same opcode may actually reference a set of microcode bits (a primitive machine instruction), a user defined procedure, or a system defined routine. The meaning depends upon the datatype and instruction descriptor for that datatype. Thus if at some time, it is decided to change the implementation of a routine, or to extend the meaning of the instruction to additional datatypes; no object code need ever be modified. One can even decide to migrate a routine into firmware without modifying any instance of its use. Because of this extensibility, the machine essentially interprets Smalltalk making compilation a simple matter of assembling opcodes. With a different set of instruction definitions it could easily be made to directly interpret APL, Prolog, SNOBOL, LISP, Backus FP, or FBAPP as well.

Abstract instruction decoding, although slow in software can be mitigated by the use of a associative mechanism in the instruction translation step which bears remarkable similarity to virtual address translation. This is an *instruction translation lookaside buffer* (ITLB), in which an opcode and the set of operand object dataypes are associated to a method.

Each ITLB corresponds to a unique method and contains three fields: 1) A *key*, containing an opcode and a set of operand classes; 2) A *primitive* bit describing whether the method is primitive or defined; and 3) A *method* field indicating how the method is to be accomplished. For example, if the primitive bit is on, the method field selects the result of a function unit. Otherwise the method field points to a piece of code defining the method.

The instruction decoding is broken down into three generic steps.

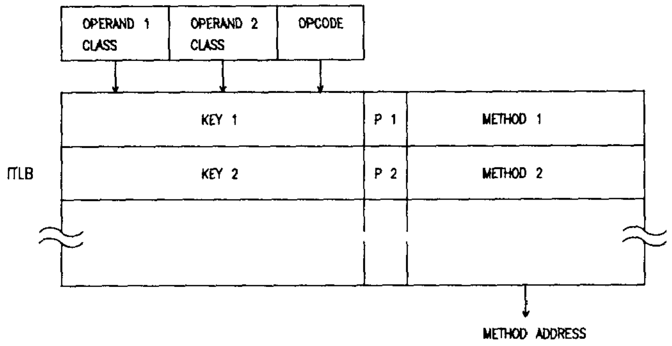

Figure 1. Instruction decoding

First, an opcode and a set of class descriptors form a key into the ITLB. An associative memory attempts to find an ITLB entry with this key. If an ITLB entry is found then the decoding proceeds. Otherwise, an instruction descriptor must be pulled in from the appropriate message dictionary, via the standard technique of method lookup (a step which always occurs in the execution of Smalltalk). Now, if the primitive bit of this entry is set then the method field sets up hardware data paths. Otherwise a procedure call is performed to an address held by the method field.

This association mechanism is pipelined with the operation of the rest of the machine. In contrast, in ordinary v.Neumann machines the association is wedged in the execution cycle, incurring an overhead on each access.

2.2 Floating Point Addresses

For an object oriented machine it is natural for an object to correspond to a single memory segment. The need to access both great numbers of small segments and a lesser number of large segments is problematical for conventional segmentation schemes. Conventional segmentation schemes divide the memory address into two fixed length fields, one of which is the segment descriptor number and the other the segment offset. The need for large numbers of segments (on the order of say a billion), demands that the segment descriptor field be much larger than is usual. On the other hand, the need for possibly large segment lengths (say a billion words), requires that the offset field also be relatively large. Current segmentation schemes choose a medium size for segments (typically 1-64K). This incurs tremendous addressing overheads for applications requiring large objects, such as image processing. On the other hand, there are far too few segments to allow allocating one per object. This dilemma is known as the small object problem.

We propose a floating point address shown as follows.

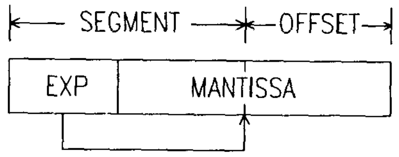

Figure 2. Floating point address

An address is given as an m bit mantissa and and e bit exponent (where $e = \lceil \lg m \rceil$). The exponent encodes the size of the offset field, shifting the binary point of the mantissa to give a *real address*. The fractional part of the real address forms the offset within the segment. The integer part of the real address when combined with the exponent names the segment descriptor. For example the 16-bit floating point address 0x8345 has an exponent of 8. Thus the offset field is the byte 0x45 and the segment number is 0x83.

Consider a comparison of floating point addressing with a typical fixed point addressing scheme: the MULTICS virtual address format. In MULTICS a 36 bit address is partitioned into two 18 bit fields. This allows 256K segments each of which may have a maximum size of 256K words. Both these limits are too restrictive for general use. This forces inappropriate grouping of small objects as well as complicated schemes to split large objects into several segments. In contrast, a 36 bit floating point address, consisting of a 5 bit exponent and 31 bit mantissa, accommodates 8 billion segments and supports segments of up to 2 billion words long. Of course segment table entries need only be kept for those segments actually allocated.

Floating point virtual addresses must be aliased when the size of an object grows out of the range of its pointer's exponents. In this event a new segment is allocated to the object and a new floating point address with a larger range than the old is allocated to the object. The segment descriptors of both the old and the new pointers are set to point to the new segment. Accesses to the object through the old segment number are allowed as long as they do not exceed the bounds set by the old exponent. When these bounds are exceeded a system trap routine replaces the old segment number with the new segment number.

2.3 Context Allocation

Contexts, activation records for Smalltalk methods, are allocated and accessed frequently: their implementation is critical to the performance of any machine. Measurements on the Smalltalk-80 system indicate that 85% of all object allocations and deallocations involve contexts [1, 19]. Thus the allocation and recycling of contexts must be made very fast while preserving the generality of possibly non-LIFO contexts. Most references are made to contexts. Measurements also show that over 91% of all memory references are to contexts [1].

We propose hardware support for allocation, deallocation, and accessing of contexts. To simplify management of the pool of free contexts, we require all contexts to be a fixed size so that a single free list can be used. Using a hardware register to point to the beginning of the free list, contexts can be allocated or freed with one memory reference.

How do we choose the length of contexts? In the COM, we chose a size of 32 words. Procedures requiring more than 32 words can allocate additional space off a heap. For C, 90% of all procedures require a frame size of fewer than 32 words [8]. Smalltalk methods tend to be much smaller than C procedures. Because of this, we believe that an overwhelming proportion of Smalltalk contexts will fit into 32 words.

Because of the generality of the Smalltalk context mechanism, strict stack based allocation and deallocation is not possible. For non-LIFO contexts, the context must be freed by a garbage collector. In current Smalltalk implementations garbage collecting consumes approximately one third of the execution time. Of this time, 82% of all allocations and deallocations occur for contexts [1]. However, 85% of contexts allocated in Smalltalk are indeed LIFO contexts and can be easily recognized [7]. These LIFO contexts are explicitly freed upon procedure exit, eliminating much of the garbage collection overhead.

Fast access to contexts is provided by a *context cache*: a set associative cache with block size equal to the context size. When a new context is allocated, it can be immediately placed in a block of the context cache and that block can be cleared. With this approach a new context does not have to be faulted in, and a free context does not have to be *cleaned* before it is reused.

Measurements indicate that most programs rarely exceed a stack depth of 1024 words or 32 contexts [8]. Thus a context cache of this modest size would almost never miss. Since the block size is large and the cache can be fairly small it is feasible to dual-port the cache by duplicating the cache directory. A dual-ported context cache can be used in place of a register file to fetch two instruction operands in parallel.

To handle larger nesting depths, a copy back mechanism could be employed to keep part of the cache free at all times. For example, when only two blocks are free in the context cache the cache begins copying the LRU context back to free additional blocks. When more than half of the cache becomes free, contexts are copied back into the cache. This copying is performed concurrently with program execution.

The context cache proposed here is very similar to the register windows used in SOAR [3] and to the stack cache proposed for the C Machine [8]. However a context cache has three significant advantages over these designs. 1) Unlike windows or stack cache, blocks in the context cache need not be contiguous. The ability to cache non-contiguous contexts is very important for non-LIFO contexts, which render the free list non-contiguous. 2) Since it associates on absolute addresses the context cache need not be invalidated on a process switch. 3) The context cache provides a mechanism to automatically initialise a new context; thus no time is wasted cleaning contexts.

§3 The Object Machine

The Caltech Object Machine (COM) is being designed as a vehicle to test the architectural features described above. The COM is designed to accelerate the execution of late binding object oriented languages.

In defining the architecture of the COM our philosophy has been to make the machine object oriented, fast, simple and flexible. The COM uses floating point addresses to give a name space which is adequate to handle many small objects and a few large objects. The memory is tagged to identify different types of objects and to allow object pointers to be used as capabilities.

Hardware support for the translation from message name to method pointer allows the COM to efficiently execute late binding object oriented languages. Speed in the COM is achieved both through the hardware method lookup and by providing hardware support for the access and allocation of contexts.

The COM is simple. All instructions are of the same length and follow the same interpretation sequence. There are no registers, all accesses are to one name space. Supporting fast access to contexts provides most of the advantages of registers without partitioning the name space or increasing the size of the processor state.

The COM achieves flexibility by providing only primitives. Higher level operating system functions such as garbage collection, process representation, and method lookup are not tied down in hardware.

3.1 Addressing

The addressing mechanism of the COM is designed to separate the issue of naming from the issue of resource allocation and to provide capability based protection for access to objects. There are three address spaces in the COM: *virtual space, absolute space, and physical space.* The issue of naming is resolved in the translation from virtual space to absolute space. The resource allocation problem is handled in the translation from absolute space to physical space.

Virtual space is a name space local to a team of processes.[4] A name within this space is a capability [9] to access an object. Virtual addresses are floating point. They may be aliased to allow teams to share objects or to allow processes within a team to access an object with different capabilities. Absolute space is the global name space. Each absolute address is a unique name identifying a particular object. All object management, for example garbage collection, is performed in absolute space. Physical space is an implementation dependent collection of storage devices each containing a number of storage locations.

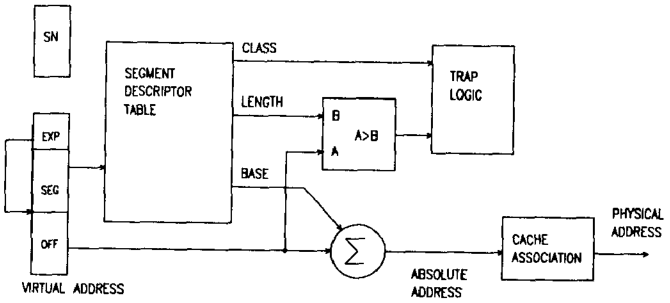

Figure 3. Address translation

The address translation mechanism of the COM is shown in figure 3 . First we translate virtual to absolute addresses to *resolve names.* The segment field and exponent field of the virtual address are concatenated to generate an index into a segment descriptor table. Each team space has its own segment descriptor table. Each entry in the segment descriptor table consists of three fields: base address, length and object class. The offset field of the virtual address is compared to the segment length field of the segment descriptor to check if the access is in bounds. If the bounds condition is met, the offset is added to the base address of the segment to generate the absolute address. All segments are aligned on absolute addresses which are multiples of their sizes so no add is required.

The COM's absolute to physical translation mechanism supports *memory resource allocation.* To translate an absolute address to a physical address the absolute address is offered to each level of the memory hierarchy in turn. Each storage device is treated as a cache in which frequently accessed portions of absolute space may be stored. This approach to resource allocation differs from the traditional use of segmentation and paging in two respects. First, absolute addresses are completely independent of the memory hierarchy. As in MULTICS [2] the same name space is used across the memory hierarchy rather than having a separate file system name space to handle slower storage devices. Also, if the mapping from absolute to physical space is performed by hashing as in a conventional set associative cache, the size of the page table is only a function of the size of physical memory and does not place a limit on the size of absolute space.

Throughout the COM, caching is used to achieve performance by accelerating frequently used translations. Because virtual addresses may be aliased and objects may move in physical memory, it is prohibitively expensive to directly cache the translation from virtual to physical space. For this reason, the translation proceeds in two steps. A virtual address is translated to an absolute address aided by an address translation lookaside buffer (ATLB). Conventional caching techniques are then used at each stage of the memory hierarchy to access physical data using an absolute address.

Virtual to absolute translations are also cached by storing directly the segment descriptors for a few frequently accessed objects. Specifically, accesses to the current method, current context, next context, and receiver are pretranslated.

3.2 Machine State

The state of the COM can be divided into a memory state and a processor state. In keeping with the goal of simplicity the processor state of the COM consists of only six registers: the context pointer (CP), the next context pointer (NCP), the free context pointer (FP), the instruction pointer (IP), the team space number (SN), and process status (PS). Only the CP needs to be saved on a method call. The CP, SN, and PS registers must be saved on a process switch.

The context pointer (CP) is a virtual address for the current context. Two locations within the context are reserved to hold the remainder of the process state: the return instruction pointer (RIP) and the return context pointer (RCP). The RIP is a virtual address which holds a continuation point in order to restart execution of a method. The IP is saved in the RIP of the current context when a method is called. Arguments are passed to a method by copying each argument into the next context before and then calling the method. When the method is called the next context becomes the current context and the method accesses its arguments as offsets from the CP. The RCP is a virtual address which points back to the calling method's context. A method returns control by reactivating this calling context.

The COM uses a tagged memory. Every word of memory has a four bit tag which is used to identify primitive types: uninitialized, small integer, floating point number, atom, instruction and object pointer. When a word is cached in the context cache, a 16-bit tag identifying the class of the object is cached with it. For primitives, this 16-bit tag is the four bit tag zero extended. For object pointers, this 16-bit tag identifies the object class and is used in the method lookup to convert an abstract instruction to a method pointer.

3.3 Instruction Set

The COM instruction set is designed to be regular so that instruction execution can be efficiently pipelined. All instructions are 32 bits in length and contain zero or three operands. Except for load and store operations all operands are referenced using one of two addressing modes. The use of three address instructions results in improved performance as the expense of larger code size. A single COM three address instruction replaces about two zero address instructions such as those used in the Smalltalk-80 Virtual Machine [11].

The two formats for COM instructions are shown below.

```
 --------------------------------
 |  O<12>  | A<6> | B<6> |  C<8> |
 --------------------------------

 --------------------------------
 |             O<31>             |
 --------------------------------
```

Figure 4. COM Instruction Formats.

Each instruction consists of an opcode, and from zero to three operands. The opcode selects which message will be performed or initiated by the instruction. The actual operation performed depends on the types of the operands. If the operands are of a type for which the machine supports a primitive method for the opcode the method will be performed directly. If no primitive operation is supported a method call will result with control being transferred to a method to perform the operation. Operands specify the data on which the instruction operates according to the addressing modes described below.

Primitive methods in the COM include:

- Arithmetic instructions: (+,-,*,/,Modulo,Negate) These instructions are defined for small integer and (except for modulo) for floating point. Some mixed mode instructions are primitive.
- Multiple precision arithmetic support: (Carry, Mult1, Mult2) These instructions, defined for small integer, allow multiple precision integer arithmetic to be implemented without flags.
- Logical and bit field instructions: (shift,arithmetic shift, rotate, mask, and, or, not, xor) Defined on small integers, these instructions treat the integers as bit fields.
- Comparisons: (<,=,=0,==) All comparisons are defined for small integer and floating point. The == (same object) comparison is defined for all types.
- Move instructions (move, movea, at:, at:put:) Move is defined for all types. Movea calculates the effective address of an object and is used to pass pointers. The at: and at:put: instructions are used to access data outside the current or next context. They are equivalent to load and store.
- Tag access: (as, tag) The *as* instruction is conditionally priviledged to prevent the forging of virtual addresses.
- Control: (fjmp,rjmp,xfer) The jump instructions jump within a method and are defined for integers objects. The xfer instruction transfers to the next context.

3.4 Addressing Modes

Two addressing modes can be used in the operand descriptors of COM instructions: context and constant. Context mode is used to access the contents of the current and next contexts. Constant mode is used to generate constants from a lookup table. The context mode uses one bit of the operand descriptor to select between the current and next contexts. The remaining bits are used as a positive offset into the context. The constant mode can only be used in the last operand descriptor of an instruction. One bit of this descriptor selects constant mode. The remaining bits index a constant table which can be used to hold frequently referenced constants including short integers, bit fields for byte insertion and the objects *true*, *false*, and *nil*.

To access outside the contexts, the at: or at:put: instructions are used. The at: instruction: `c0 <- c1 at: c2` fetches `c0` from the field indexed by `c2` in the object pointed at by `c1`. The at:put: instruction: `c1 at: c2 put c0` reverses this operation. These instructions provide indexed addressing, or if `c2` is replaced by a constant, displacement addressing. Because memory access is restricted to these two instructions, the COM pipeline rarely has to wait for a memory cycle to complete.

3.5 Method Call

When an instruction with an unimplemented opcode is executed or when an instruction is executed on operands which are not the type for which the instruction is implemented (or mixed), control is transferred to the method which implements the proper operation.

The method to be executed is determined from the opcode and operands types by a lookup in the ITLB described above. For the sero operand instruction, sero, one or two locals in the next context are considered as operands depending on the high order bits of the instruction.

A new 32-word context is allocated for each method call. This context is allocated in advance so that arguments can be passed to a method by copying them into the new context before performing the method call. This argument passing is performed automatically for non-primitive methods which are formatted as one two or three operand instructions. For these instructions the processor expands the operands into words and copies them to the new context. For methods which do not include operands in their instruction formats, the programmer must place arguments in the next context. Note that copying arguments is not in general required. A three address instruction can place the result of an operation directly into the next context. When a method completes it is expected to place its result (if any) at the address specified by the first operand and to return control to the calling method by executing an instruction with the return bit set.

3.6 Implementation

Caching and pipelining are used to achieve performance in the design of the COM. A block diagram of a proposed COM design in shown in figure 5 . Caching is used in four places in the processor.

- The CP, NCP, and IP are pre-translated to absolute addresses and are cached in special hardware registers.
- An instruction cache holds the instructions of frequently accessed methods.
- An instruction translation lookaside buffer (ITLB) holds associations from message name and argument type to method pointer.
- Finally a context cache is used to cache recently accessed contexts. This cache makes context accesses as fast as register accesses. We describe the context cache below.

Instruction interpretation proceeds in five steps. Each block and signal in the block diagram is labeled with the step during which it is active.

1. The instruction pointer is used to lookup the next instruction in the instruction cache.
2. The operands and their tags are fetched. Operand fetches will be from either the context cache or a constant generator.
3. The opcode of the instruction and the types of the operands are translated by the ITLB into either a bit vector describing a primitive operation or a method pointer to be used in a method call.
4. For primitive methods, the operation is performed. Note that since the operands were available at the end of step 2, a dedicated function unit has two steps to compute a result.
5. The results of the operation are stored and the IP is incremented.

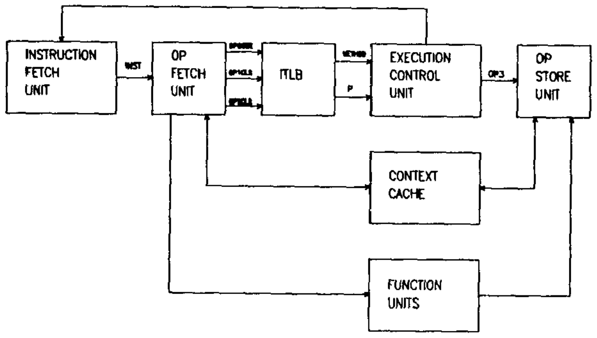

Figure 5. COM Block Diagram.

This instruction interpretation sequence can be pipelined as shown below so that a new instruction is started every two clock cycles. This instruction rate is limited by the context cache. It is assumed that the cache can perform two reads or one write each cycle, but cannot perform reads and writes simultaneously. This pipelining is shown below. Since the $i+1^{st}$ instruction reads its operands before the i^{th} instruction has written its result, an interlock is required. Also, the pipeline may be stalled by a miss in any cache, or by an at: or at:put: instruction. A branch instruction is delayed one clock cycle as in the MIPS processor [12] and does not interfere with operation of the pipeline. Also following the example of MIPS, the compiler is required to assure that an instruction does not attempt to read the result of the previous instruction eliminating the need for an interlock or bypass. A non primitive method is detected in step three, flushes the next instruction which has already been fetched and initiates the method call sequence described below.

```
Instr1

+--------+
| Fetch  |
+--------+  Instr2
| Read   |
+--------+  +--------+
| ITLB   |  | Fetch  |
+--------+  +--------+  Instr3
| Op     |  | Read   |
+--------+  +--------+  +--------+
| Write  |  | ITLB   |  | Fetch  |
+--------+  +--------+  +--------+
            | Op     |  | Read   |
            +--------+  +--------+
            | Write  |  | ITLB   |
            +--------+  +--------+
                        | Op     |
                        +--------+
                        | Write  |
                        +--------+
```

Figure 6. Instruction Pipelining.

Detection of a method call during step three causes the following operations to be performed. Except for the copying of operands into the new context, all of these operations can be performed in one clock cycle. Only the IP needs to be saved in the current context. The current CP was saved in the next context when it was created. Thus, a method call with no operands only delays execution four clock cycles: two to execute the instruction which caused the call, one for flushing the instruction in the pipeline, and one for performing the operations listed below. An additional cycle is required for each operand copied to the next context.

- Flush pipeline of next instruction and roll back IP to instruction following call.
- Store IP into the context.
- CP ← NCP. Note that CP is already stored as RCP in the next context.
- Initiate the allocation of a new context. Any NCP relative accesses will be held up until the new context is available.
- Set IP to point to the first instruction of the new method.
- If not a zero operand instruction copy operands into new context.

A return instruction reverses these operations by setting the CP ← RCP, and restoring the IP from the caller's context. Since return can be detected early in the pipeline it can be processed with no delay. Thus method returns cost only two clock cycles.

A critical component of the architecture is the context cache. Like register windows in RISC and SOAR [3] and the C machine stack cache [8], we take advantage of the fact that caches can be made to operate as fast as registers. A block diagram of an implementation of the context cache is shown below:

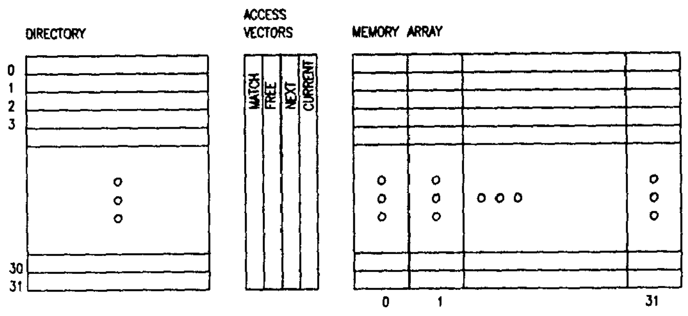

Figure 7. Context Cache Block Diagram.

The Context Cache consists of two parts: the directory and the data memory. Our scheme achieves speed by bypassing the directory on accesses to the current or next context. The directory is an associative memory with an entry for each block of the context cache. Each entry holds the absolute address of the context cached in the corresponding block. The data memory consists of four 32 bit *access vectors* and a dual port memory array consisting of 32 blocks of 32 words each (each block holds one context). Special circuitry in the memory array permits an entire block to be cleared in a single operation. The access vectors facilitate fast access to and allocation of the memory array. Each access vector is a bit vector specifying a set of blocks in the cache.

The four vectors are:

1. The *current vector* specifies a singleton set containing the current context.
2. The *next vector* specifies a singleton set containing the next context.
3. The *free vector* encodes the set of blocks which are currently unused.
4. The *match vector* specifies a singleton set containing the context associated with an absolute address match in the directory.

There are two methods for accessing the cache. The current and next access vectors are used to provide fast access the current and next context by immediately selecting the correct block. A five bit address is used to select the word within the block to be read or written. To access a context using an absolute address, the address is input to the cache directory. If it matches a directory entry, the corresponding bit in the match vector is set. The match vector is then used to access the memory array.

To allocate a new context as the next context, the first free bit of the free vector is set to zero and the corresponding bit of the next vector is set to one. The new context is then cleared, and the absolute address is written into the directory.

On a method call, the next vector is moved to the current vector and a new next context is allocated as described above. On return from a method, the current vector is moved back to the next vector and an association in the directory is used to set the current vector.

§4 The Smalltalk Execution model

In this section we discuss how the Smalltalk execution model is adapted to our architecture. The Smalltalk virtual machine is considerably different from our proposal. It is a zero instruction stack machine. Its context is a small fixed length object which has links to a variable length expression stack as well as a number of other links to contexts and objects. COM has no expression stack and uses three address instructions.

Of the four objects that can be directly addressed through "registers", three are used by the Smalltalk compiler. They are the *self* object, the *current context object* , and the *next context object.* The *receiver object* allows quick access to the fields of the object which is the receiver of the current message. The *current context* holds all the information needed for executing the current method. The *next context* is used to set up procedure linkage and argument transmission for a message to be called.

Methods are of two types: primitive methods, and defined methods. A primitive method takes its arguments and results directly from a machine instruction, so it doesn't need a context. Nonprimitive methods need to be able to bind their arguments and results so a context must be set up for them.

For a nonprimitive message send we first set up the context. Which appears in figure 8.

```
---------------------------------------------
  Context:
       RCP  (link to sending context).
       RIP  (encodes the method and offset).
       arg0 (where to store the result).
       arg1 (receiver of message).
       arg2
         .
         .
       argN
       temp1
         .
         .
       tempN
---------------------------------------------
```

Figure 8. A Smalltalk context.

The first field of a context is a link to the sending context. This link is filled in when a context immediately upon creation by the procedure linkage sequence. The next field is the RIP. This field holds a pointer into the method being executed by this context. Note that the pointer encodes both the method object and the offset within the method. The RIP field is initialized at context creation and is updated each time the context transfers control to another context.

Argument 0 points to a location to store the returned value of a method. The receiver of the message is argument 1. And further arguments of the method are stored in successive locations, determined at compile time. All temporaries are then stored in the remaining locations. A temporary may be a local variable for the method or may arise from expression evaluation, since we forego the use of an expression stack.

To call a method the compiler generates code to load arguments in the next context, to fill in the result pointer and to transfer to the next context. Because the method indirects through the result pointer, argument transmission is quite flexible. Here is an example:

```
---------------------------------------------------
foo | | ^self * (self-1) bar.

 n1=c1-1              ; self-1
 n0=&c2               ; effective addr of c2 to n0.
 bar                  ; Call bar.
 c2=c1*c2             ; Compute the product.
 *c0=c2   (return)    ; Return the result
---------------------------------------------------
```

Figure 9. Example of compiled code.

§5 Experimental Results

This section presents the results of experiments run to test the utility of hardware support for method lookup. These experiments were run on an simulator of an early version of the COM called the Fith Machine. The Fith Machine was motivated by the Fith programming language.[13] The Fith language combines the syntax of Forth with the semantics of smalltalk. Since Fith is a stack based language, the Fith Machine was a stack machine and had an instruction set very different from the three address instruction set of the COM; however the instruction translation mechanisms of the two machines are identical so the results presented here should apply to the COM as well.

The experiments were run on the Fith Machine simulator, a suite of C programs including a Fith interpreter and a cache simulator which processed address traces to produce cache statistics. Traces of large Fith programs were produced by instrumenting the Fith interpreter on an IBM 4341 to record for each instruction interpreted: the address of the instruction, the opcode, and the type of object on the top of the stack. Several traces were produced, the longest of which was about 20,000 instructions in length. For each trace, the instruction cache hit ratio and ITLB hit ratio was recorded for several cache sizes and associativities. A warmup trace was run before the measurement trace to avoid biasing the results by the initial faulting in of data into the caches.

The results of these simulations are shown in figures 10 and 11. The hit ratio in the ITLB for cache sizes varying from 8 to 4096 is shown in figure 10. The data indicate that a 99% hit ratio can be realized with a 512 entry 2-way associative cache. It is interesting to note that the data for one-way associative or direct mapped caches in figure 10 agree within a few percent with data published on the performance of a direct mapped software cache in the Berkeley Smalltalk system. [5] It is clear from the figure that a great deal can be gained by having at least a 2-way associative cache. It is not clear that adding more associativity improves the hit ratio much.

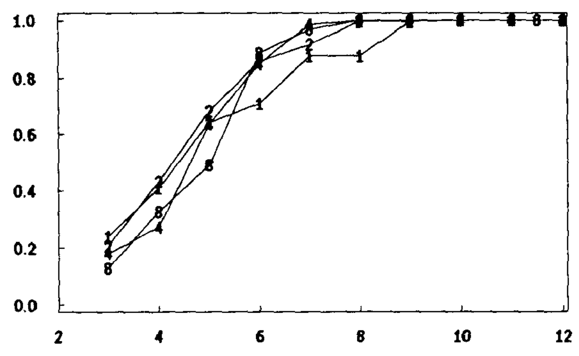

Figure 10. ITLB Hit Ratio vs. Log₂ of Cache Size

The hit ratio in the instruction cache is shown in figure 11 for cache sizes varying from 8 to 4096. In this case it appears that a 2 or 4-way associative cache with 4096 entries is required to achieve a 99% hit ratio.

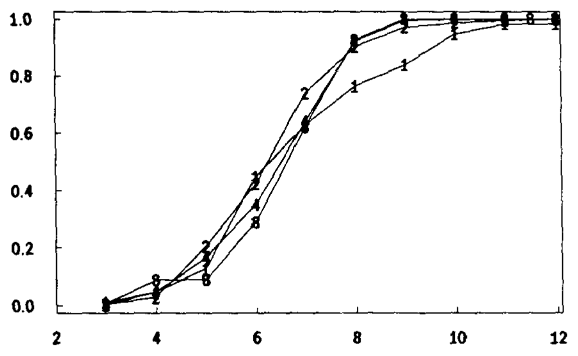

Figure 11. Instruction Cache Hit Ratio vs. $\log_2$ of Cache Size

These experiments verified the validity of using an ITLB to support run time binding of methods to messages. A modest size ITLB results in an acceptable hit ratio of 99% . If this hit ratio is insufficient, a larger second level ITLB can be implemented in main memory and accessed by miss processing hardware. Only a miss in both caches would result in a trap.

It was during the course of these experiments that the architecture of the Fith Machine was dropped and the COM architecture was defined. Stack machines while offering small code size require almost twice as many instructions to implement a given source language program than a three address machine. Our initial design studies indicated that executing a stack machine instruction would take about the same amount of time as executing a three address instruction. From this analysis, the three address COM should offer a significant performance improvement over a stack machine.

Another decision which came out of these experiments was to abandon the stack based control structure of the Fith Machine in favor of the more general contexts of the COM. The contexts in COM support a general control transfer similar to Lampson's XFER instruction.[15] This control transfer supports block contexts in Smalltalk, process switch, and interrupts.

§6 Conclusion

Late binding object oriented languages need not be slow; the inclusion of a modest amount of hardware support can improve the performance of these languages making them competitive with conventional languages. In the past, the well known software engineering advantages of object oriented languages have been restricted to applications for which speed is not an issue. With the improved performance provided by our proposed hardware features these languages may be employed in a wider range of real world applications.

We have proposed the following novel concepts: abstract instructions, floating point addresses, three level addressing and hardware support for contexts. Abstract instructions with translation lookaside buffering efficiently implement the semantics of method lookup. Floating point addresses solve the small object problem. Three level addressing separates the issue of naming from that of resource allocation in a memory hierarchy. Providing fast access to contexts combines the speed advantages of registers without sacrificing the simplicity of a single virtual name space. Accelerating allocation of arbitrary non-LIFO contexts reduces the burden of storage management.

We have done the following: A simulator has been written to test the use of lookaside buffering in accelerating interpretation of abstract instructions. Based on the results of these experiments we have defined the architecture of the Caltech Object Machine embodying the above ideas. A Smalltalk-80 compiler has been written which generates code for the COM. A function block level simulator of the COM is under construction. We plan to build a prototype COM using both catalog parts and custom integrated circuits.

References

[1] Baden, Scott B., "Low-Overhead Storage Reclamation in the Smalltalk-80 Virtual Machine," in *Smalltalk-80, Bits of History, Words of Advice,* Glenn Krasner, ed., Addison-Wesley, 1983, pp. 331-342.

[2] Bensoussan, A. et. al., "The Multics Virtual Memory: Concepts and Design," *Comm. ACM 15,5,* (May 1972), pp. 308-318.

[3] Blakkan, John, "Register Windows for SOAR," in *Proceedings of CS292R: Smalltalk on a RISC, Architectural Investigations,* Computer Science Division, University of California, Berkeley, April, 1983, pp. 126-140.

[4] Cheriton, David R. et. al., "Thoth, a Portable Real-Time Operating System," *Comm. ACM 22,2,*(February 1979), pp. 105-115.

[5] Conroy, T.J. and Pelegri-Llopart, Eduardo, "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations," in *Smalltalk-80, Bits of History, Words of Advice,* Glenn Krasner, ed., Addison-Wesley, 1983, pp. 239-247.

[6] Deutsch, L. Peter, "The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture," in *Smalltalk-80, Bits of History, Words of Advice,* Glenn Krasner, ed., Addison-Wesley, 1983, pp. 189-206.

[7] Deutsch, L. Peter, and Schiffman, Allan M., *Efficient Implementation of the Smalltalk-80 System,* Fairchild Technical Report No. 651, January 1984.

[8] Ditzel, David R. and McLellan H.R., "Register Allocation for Free: The C Machine Stack Cache," *Proceedings ACM Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, pp. 48-56.

[9] Fabry, R.S., "Capability Based Addressing," *Comm. ACM 17,7,* (July 1974), pp. 403-412.

[10] Falcone, Joseph R., "The Analysis of the Smalltalk-80 System at Hewlett-Packard," in *Smalltalk-80, Bits of History, Words of Advice,* Glenn Krasner, ed., Addison-Wesley, 1983, pp. 207-237.

[11] Goldberg, Adele and Robson, David, *Smalltalk-80, The Language and its Implementation,* Addison-Wesley, 1983, pp. 567-591.

[12] Hennessy, John, et. al., "Hardware/Software Tradeoffs for Increased Performance," *Proceedings ACM Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, pp. 2-11.

[13] Kajiya, James T. and Draper D.D. *FITH Language Reference Manual,* Caltech Computer Science Display File. 1983.

[14] Krasner, Glenn, ed., *Smalltalk-80, Bits of History, Words of Advice,* Addison-Wesley, 1983.

[15] Lampson, Butler W., "Fast Procedure Calls," *Proceedings ACM Symposium on Architectural Support for Programming Languages and Operating Systems,* 1982, pp. 66-76.

[16] Norton, Richard L., and Abram, Jacob A., "Adaptive Interpretation as a Means of Exploiting Complex Instruction Sets," *10th ACM Symposium on Computer Architecture,* 1983, pp. 277-282.

[17] Patterson, David A., ed. *Proceedings of CS292R: Smalltalk on a RISC, Architectural Investigations,* Computer Science Division, University of California, Berkeley, April, 1983.

[18] Sites, Richard L., "How to use 1000 Registers," *Proceedings of Caltech Conference on VLSI,* January, 1979, pp 527-532.

[19] Ungar, David M. and Patterson, David A., "Berkeley Smalltalk: Who Knows Where the Time Goes," in *Smalltalk-80, Bits of History, Words of Advice,* Glenn Krasner, ed., Addison-Wesley, 1983, pp. 189-206.